Ecommerce Database Management System

This project is developed as part of the University PTU Curriculum for the Database Management Systems (DBMS) course. It focuses on designing and implementing a **robust, efficient, and feature-rich e-commerce database** using PostgreSQL. The project simulates real-world online shopping platforms and aims to **facilitate small sellers transitioning from offline to online commerce** through a scalable and consistent database infrastructure.

The database is built with a strong focus on:

- **Data consistency and integrity**
- **Efficient query execution**
- **Support for transactional operations**
- **Ease of use for customers and sellers**

To further enhance usability, this system also incorporates **advanced PostgreSQL features** such as triggers, stored procedures, derived attributes, and rating-based queries for product recommendations.

Project Description

In the modern era of digital commerce, businesses increasingly seek to move away from traditional offline selling toward online platforms. A critical enabler of this transition is a well-structured and optimized database system that ensures reliable storage, efficient data access, and smooth transactional operations.

This project focuses on building a **comprehensive e-commerce database** that supports a variety of operations including:

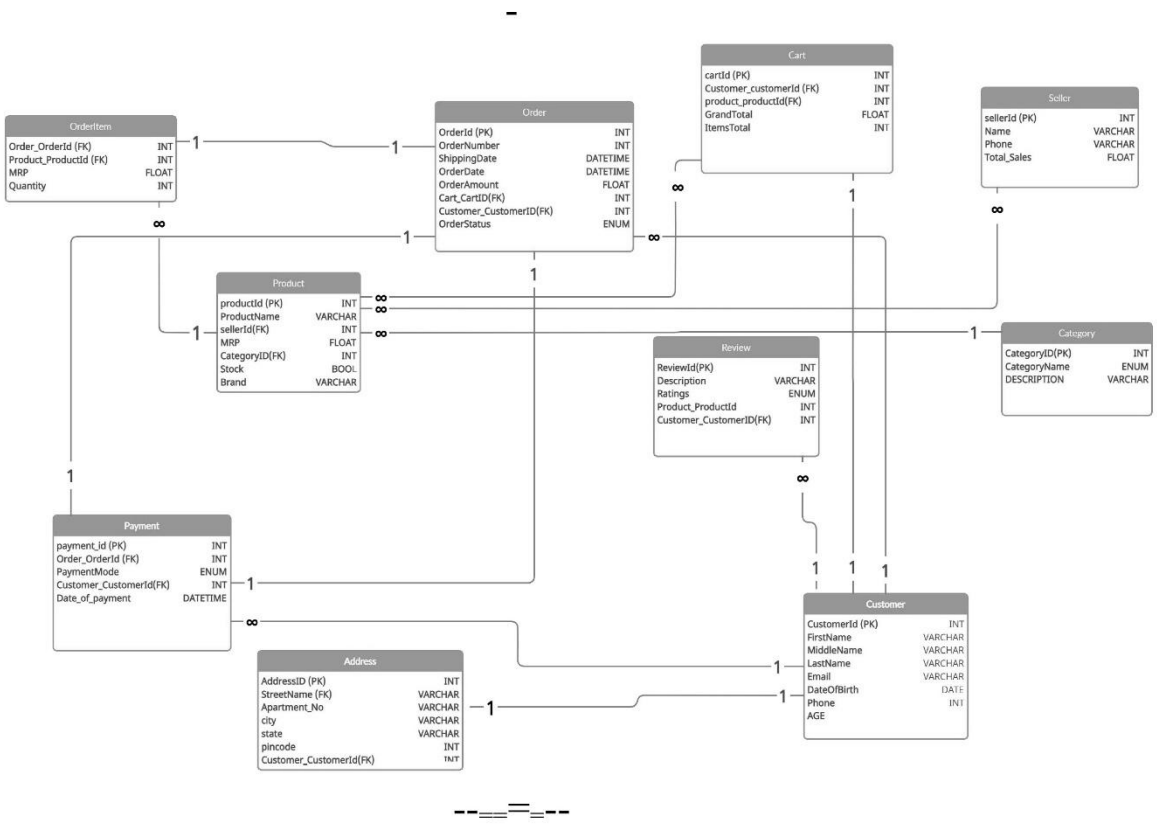
- Viewing and placing customer orders
- Maintaining and updating user and product data
- Managing cart items and calculating totals
- Tracking order status and payment details
- Recording and analyzing customer reviews
- Ensuring data accuracy through constraints and triggers

The goal is to design a solution that **mirrors the functional and performance needs** of real-world online marketplaces while also offering enhanced analytical capabilities, such as **identifying top-rated products, most active sellers, and customer preferences**

Functional Requirements

- Customers can view and update their personal account details.
- Customers can browse and search products by category.
- Customers can add products to their cart or wishlist and view the total amount.
- Customers can update cart contents by adding or removing products.
- Customers can choose different payment modes (COD, UPI, Card, etc.).
- Customers can track the status of their orders post-purchase.
- Customers can rate and review products they've purchased.
- Sellers can update stock availability for their listed products.
- Sellers can monitor their total sales on a daily, monthly, or yearly basis.
- Customers and sellers have isolated access—customers cannot view seller details and vice versa.
- The system ensures data consistency and transactional integrity to prevent data loss.
- Triggers are used to automatically update stock and order amounts.
- Analytical queries are included to support sales insights and product recommendations.

Relational Database Schema



The database schema includes entities and their relationships to support e-commerce operations. Below is the schema with entities, attributes, and their types.

Entities and Attributes

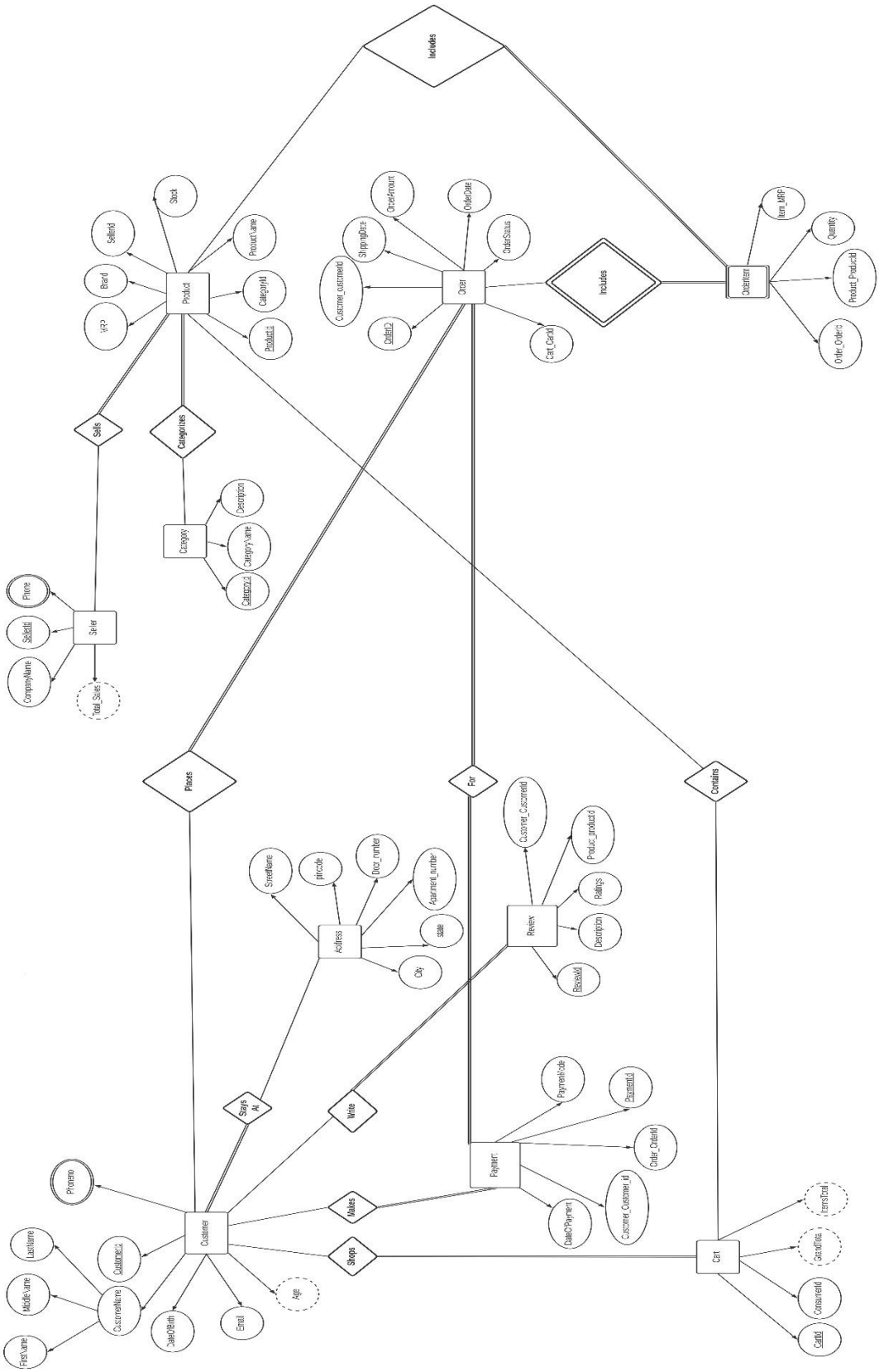
ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
Customer	Customer_CustomerId Name Email DateOfBirth Phone Age	Simple Composite Simple Simple Multivalued Derived	Strong
Order	OrderId ShippingDate OrderDate OrderAmount Cart_CartID	Simple Simple Simple Simple Simple	Strong
OrderItem	Order_OrderId (PK) Product_ProductId(FK) MRP Quantity	Simple Simple Simple Simple	Weak
Product	productId (PK) ProductName sellerId MRP CategoryID Stock Brand	Simple Simple Simple Simple Simple Simple Simple	Strong
Review	ReviewId(PK) Description Ratings Product_ProductId Customer_CustomerID	Simple Simple Simple Simple	Strong
Cart	cartId (PK) Customer_customerId(FK)	Simple Simple	Strong

ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
	GrandTotal ItemsTotal	Derived Derived	
Category	CategoryID(PK) CategoryName DESCRIPTION	Simple Simple Simple	Strong
seller	sellerId (PK) Name Phone Total_Sales	Simple Simple Multivalued Derived	Strong
Payment	payment_id Order_OrderId PaymentMode Customer_CustomerId PaymentDate	Simple Simple Simple Simple Simple	Strong

Entity Relationships and Cardinality

ENTITIES	RELATION	CARDINALITY	TYPE OF PARTICIPATION
Customer Address	Stays At	OneToOne	Total Partial
Customer Cart	Shops	OneToOne	Partial Total
Customer Order	Places	OneToMany	Partial Total
Customer Payment	Makes	OneToMany	Partial Total
Customer Review	Write	OneToMany	Partial Total
Seller Product	Sells	ManyToMany	Partial Total
Category Product	Categorizes	OneToMany	Partial Total
Cart Product	Contains	ManyToMany	Partial Partial
Product OrderItem	Includes	OneToMany	Partial Total
Order OrderItem	Includes	OneToOne	Partial Total
Payment Order	For	OneToOne	Total Total

ER-Diagram


Includes
Product
Stock
SellerId
ProductName
Brand
CategoryId
MRP
ProductId
Sells
Categorizes
Category
Description
CategoryName
CategoryId
Seller
Phone
SellerId
CompanyName
Total_Sales
Order
OrderAmount
OrderDate
ShippingDate
OrderStatus
Customer_customerId
OrderID
Cart_CartId
Includes
OrderItem
Item_MRP
Quantity
Product_ProductId
Order_OrderID
Places
For
Contains
Address
StreetName
pincode
Door_number
Apartment_number
state
City
Review
Customer_CustomerId
Product_productId
Ratings
Description
ReviewId
Stays At
Write
Payment
PaymentMode
PaymentId
Order_OrderId
Customer_Customer_id
Date&Payment
Makes
Shops
Cart
ItemsTotal
GrandTotal
ConsumerId
CartId
Customer
Phonenno
LastName
CustomerName
MiddleName
FirstName
DateOfBirth
Email
Age

DDL Queries

Below are the Data Definition Language (DDL) queries to create the database and tables using PostgreSQL syntax.

```
-- Create Database
CREATE DATABASE ecommerce_db;
\c ecommerce_db;

-- Create Customer Table
CREATE TABLE customer (
    customer_id INTEGER PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100) UNIQUE,
    date_of_birth DATE,
    phone VARCHAR(15),
    age INTEGER GENERATED ALWAYS AS (DATE_PART('year', AGE(date_of_birth))) STORED
);

-- Create Address Table
CREATE TABLE address (
    address_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    street VARCHAR(100),
    city VARCHAR(50),
    pincode VARCHAR(10),
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

```
-- Create Cart Table
CREATE TABLE cart (
    cart_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

-- Create Category Table
CREATE TABLE category (
    category_id INTEGER PRIMARY KEY,
    name VARCHAR(50)
);

-- Create Product Table
CREATE TABLE product (
    product_id INTEGER PRIMARY KEY,
    name VARCHAR(100),
    price NUMERIC(10,2),
    stock INTEGER,
    category_id INTEGER,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- Create Seller Table
CREATE TABLE seller (
    seller_id INTEGER PRIMARY KEY,
    name VARCHAR(100)
```

```
);

-- Create Order Table
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    order_date DATE,
    shipping_date DATE,
    order_amount NUMERIC(10,2),
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

-- Create Payment Table
CREATE TABLE payment (
    payment_id INTEGER PRIMARY KEY,
    order_id INTEGER,
    payment_mode VARCHAR(20) CHECK (payment_mode IN ('COD', 'CreditCard', 'DebitCard',
'UPI')),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

-- Create Review Table
CREATE TABLE review (
    review_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    product_id INTEGER,
    rating INTEGER CHECK (rating BETWEEN 1 AND 5),
    comment TEXT,
```

```
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create OrderItem Table
CREATE TABLE order_item (
    order_item_id INTEGER PRIMARY KEY,
    order_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create Seller_Product (Many-to-Many relationship between Seller and Product)
CREATE TABLE seller_product (
    seller_id INTEGER,
    product_id INTEGER,
    PRIMARY KEY (seller_id, product_id),
    FOREIGN KEY (seller_id) REFERENCES seller(seller_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create Cart_Product (Many-to-Many relationship between Cart and Product)
CREATE TABLE cart_product (
    cart_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
```

```
    PRIMARY KEY (cart_id, product_id),
    FOREIGN KEY (cart_id) REFERENCES cart(cart_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

DML Queries

Insert Queries

Below are sample INSERT queries to populate the database with initial data.

```
-- Insert Customers
INSERT INTO customer (customer_id, first_name, last_name, email, date_of_birth, phone)
VALUES
  (1, 'John', 'Doe', 'john.doe@email.com', '1990-05-15', '1234567890'),
  (2, 'Jane', 'Smith', 'jane.smith@email.com', '1985-08-22', '0987654321');

-- Insert Addresses
INSERT INTO address (address_id, customer_id, street, city, pincode)
VALUES
  (1, 1, '123 Main St', 'New York', '10001'),
  (2, 2, '456 Oak Ave', 'Los Angeles', '90001');

-- Insert Categories
INSERT INTO category (category_id, name)
VALUES
  (1, 'Electronics'),
  (2, 'Clothing');

-- Insert Products
INSERT INTO product (product_id, name, price, stock, category_id)
VALUES
  (1, 'Smartphone', 699.99, 50, 1),
  (2, 'T-Shirt', 19.99, 100, 2);

-- Insert Sellers
```

```
INSERT INTO seller (seller_id, name)
VALUES
   (1, 'TechTrend Innovations'),
   (2, 'FashionHub');

-- Insert Seller_Product Relationships
INSERT INTO seller_product (seller_id, product_id)
VALUES
   (1, 1),
   (2, 2);

-- Insert Carts
INSERT INTO cart (cart_id, customer_id)
VALUES
   (1, 1),
   (2, 2);

-- Insert Cart_Product Relationships
INSERT INTO cart_product (cart_id, product_id, quantity)
VALUES
   (1, 1, 2),
   (2, 2, 3);

-- Insert Orders
INSERT INTO orders (order_id, customer_id, order_date, shipping_date, order_amount)
VALUES
   (1, 1, '2025-06-01', '2025-06-05', 1399.98),
   (2, 2, '2025-06-02', '2025-06-06', 59.97);
```

```
-- Insert OrderItems
INSERT INTO order_item (order_item_id, order_id, product_id, quantity)
VALUES
    (1, 1, 1, 2),
    (2, 2, 2, 3);

-- Insert Payments
INSERT INTO payment (payment_id, order_id, payment_mode)
VALUES
    (1, 1, 'CreditCard'),
    (2, 2, 'COD');

-- Insert Reviews
INSERT INTO review (review_id, customer_id, product_id, rating, comment)
VALUES
    (1, 1, 1, 5, 'Excellent smartphone!'),
    (2, 2, 2, 4, 'Comfortable T-shirt.');
```

Sample Queries

Below are example DML queries to demonstrate key operations based on the provided requirements, written in PostgreSQL.

1. **Find products with the highest ratings for a given category (Category: Electronics)**

```
SELECT p.product_id, p.name, AVG(r.rating)::NUMERIC(3,2) AS avg_rating
FROM product p
JOIN review r ON p.product_id = r.product_id
WHERE p.category_id = 1
GROUP BY p.product_id, p.name
ORDER BY avg_rating DESC
LIMIT 5;
```

---

2. **Filter products by brand and price (Price < 100)**

```
SELECT p.product_id, p.name, p.price, s.name AS seller_name
FROM product p
JOIN seller_product sp ON p.product_id = sp.product_id
JOIN seller s ON sp.seller_id = s.seller_id
WHERE p.price < 100
ORDER BY p.price;
```

---

3. **Calculate total price in a customer's cart**

```
SELECT c.cart_id, SUM(p.price * cp.quantity) AS total_amount
FROM cart c
JOIN cart_product cp ON c.cart_id = cp.cart_id
JOIN product p ON cp.product_id = p.product_id
```

```
WHERE c.customer_id = 1
GROUP BY c.cart_id;
```

---

4. **Find the best seller for a particular product** (ProductID: 1)

```
SELECT s.seller_id, s.name, COUNT(oi.order_id) AS total_orders
FROM seller s
JOIN seller_product sp ON s.seller_id = sp.seller_id
JOIN order_item oi ON sp.product_id = oi.product_id
WHERE sp.product_id = 1
GROUP BY s.seller_id, s.name
ORDER BY total_orders DESC
LIMIT 1;
```

---

5. **List orders to be delivered at a particular pincode**

```
SELECT o.order_id, o.order_date, o.order_amount
FROM orders o
JOIN customer c ON o.customer_id = c.customer_id
JOIN address a ON c.customer_id = a.customer_id
WHERE a.pincode = '10001';
```

---

6. **List products with the highest sales on a particular day** (Date: 2025-06-01)

```
SELECT p.product_id, p.name, SUM(oi.quantity) AS total_sold
FROM product p
JOIN order_item oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
```

```
WHERE o.order_date = '2025-06-01'
GROUP BY p.product_id, p.name
ORDER BY total_sold DESC
LIMIT 1;
```

---

7. **List categories with the highest sales on a particular day** (Date: 2025-06-01)

```
SELECT c.category_id, c.name, SUM(oi.quantity) AS total_sold
FROM category c
JOIN product p ON c.category_id = p.category_id
JOIN order_item oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.order_date = '2025-06-01'
GROUP BY c.category_id, c.name
ORDER BY total_sold DESC
LIMIT 1;
```

---

8. **List customers who bought the most from a particular seller**

```
SELECT c.customer_id, c.first_name, c.last_name, COUNT(o.order_id) AS total_orders
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN seller_product sp ON oi.product_id = sp.product_id
WHERE sp.seller_id = 1
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_orders DESC
LIMIT 5;
```

---

9. **List orders with non-COD payment modes that are yet to be delivered**

```
SELECT o.order_id, o.order_date, p.payment_mode
FROM orders o
JOIN payment p ON o.order_id = p.order_id
WHERE p.payment_mode != 'COD' AND o.shipping_date > CURRENT_DATE;
```

---

10. **List orders with total amount greater than 5000**

```
SELECT o.order_id, o.order_date, o.order_amount
FROM orders o
WHERE o.order_amount > 5000;
```

---

11. Product Recommendation Queries

```
SELECT oi2.product_id, p.name, COUNT(*) AS purchase_count
FROM order_item oi1
JOIN order_item oi2 ON oi1.order_id = oi2.order_id AND oi1.product_id != oi2.product_id
JOIN product p ON p.product_id = oi2.product_id
WHERE oi1.product_id = 1
GROUP BY oi2.product_id, p.name
ORDER BY purchase_count DESC
LIMIT 5;
```

---

12. **Popular Products in a Category (Based on Reviews)**

```
SELECT p.product_id, p.name, AVG(r.rating) AS avg_rating, COUNT(r.rating) AS total_reviews
FROM product p
JOIN review r ON r.product_id = p.product_id
WHERE p.category_id = 1 -- replace with desired category
```

```
GROUP BY p.product_id, p.name
HAVING COUNT(r.rating) >= 2
ORDER BY avg_rating DESC
LIMIT 5;
```

---

13. **Time-Based Sales Analytics: Best-Selling Products Last Month**

```
SELECT p.product_id, p.name, SUM(oi.quantity) AS units_sold
FROM product p
JOIN order_item oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.order_date >= date_trunc('month', CURRENT_DATE) - INTERVAL '1 month'
  AND o.order_date < date_trunc('month', CURRENT_DATE)
GROUP BY p.product_id, p.name
ORDER BY units_sold DESC
LIMIT 5;
```

---

14. **Top Customers by Spending**

```
SELECT c.customer_id, c.first_name, c.last_name, SUM(o.order_amount) AS total_spent
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spent DESC
LIMIT 5;
```

---

15. **Stored Procedure: Top Sellers by Category**

```
CREATE OR REPLACE PROCEDURE top_sellers_by_category(cat_id INTEGER)
LANGUAGE plpgsql
AS $$
BEGIN
   SELECT s.seller_id, s.name, COUNT(oi.order_id) AS total_orders
   FROM seller s
   JOIN seller_product sp ON s.seller_id = sp.seller_id
   JOIN product p ON p.product_id = sp.product_id
   JOIN order_item oi ON oi.product_id = p.product_id
   WHERE p.category_id = cat_id
   GROUP BY s.seller_id, s.name
   ORDER BY total_orders DESC
   LIMIT 5;
END;
```

---

16. **Detect and List Inactive Customers**

```
SELECT c.customer_id, c.first_name, c.last_name
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id
   AND o.order_date >= CURRENT_DATE - INTERVAL '60 days'
WHERE o.order_id IS NULL;
```

---

17. **Frequent Co-Purchased Category Pairings**

```
SELECT c1.name AS category1, c2.name AS category2, COUNT(*) AS co_purchases
FROM order_item oi1
```

```
JOIN product p1 ON oi1.product_id = p1.product_id
JOIN category c1 ON p1.category_id = c1.category_id

JOIN order_item oi2 ON oi1.order_id = oi2.order_id AND oi1.product_id != oi2.product_id
JOIN product p2 ON oi2.product_id = p2.product_id
JOIN category c2 ON p2.category_id = c2.category_id

GROUP BY c1.name, c2.name
ORDER BY co_purchases DESC
LIMIT 10;
```

Trigger: Update Stock After Payment

```
CREATE OR REPLACE FUNCTION update_stock_after_payment()
RETURNS TRIGGER AS $$
BEGIN
   UPDATE product
   SET stock = stock - oi.quantity
   FROM order_item oi
   WHERE oi.product_id = product.product_id
   AND oi.order_id = NEW.order_id;
   RETURN NEW;
END;

CREATE TRIGGER update_stock_trigger
AFTER INSERT ON payment
FOR EACH ROW
EXECUTE FUNCTION update_stock_after_payment();
```

Trigger: Update Order Amount

```
CREATE OR REPLACE FUNCTION update_order_amount()
RETURNS TRIGGER AS $$
BEGIN
   UPDATE orders
   SET order_amount = (
      SELECT SUM(p.price * oi.quantity)
      FROM order_item oi
      JOIN product p ON oi.product_id = p.product_id
      WHERE oi.order_id = NEW.order_id
```

```
    )
    WHERE order_id = NEW.order_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_order_amount_trigger
AFTER INSERT ON order_item
FOR EACH ROW
EXECUTE FUNCTION update_order_amount();
```

Trigger: Auto-Insert Positive Feedback if Rating ≥ 4

```
CREATE OR REPLACE FUNCTION add_auto_comment()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.comment IS NULL AND NEW.rating >= 4 THEN
        NEW.comment := 'Thanks for the great rating!';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER auto_feedback_trigger
BEFORE INSERT ON review
FOR EACH ROW
EXECUTE FUNCTION add_auto_comment();
```